Introduction to Java Pathfinder

Cyrille Artho
KTH Royal Institute of Technology, Stockholm, Sweden
School of Electrical Engineering and Computer Science
Theoretical Computer Science
artho@kth.se

Today's objectives

1. Model Checking vs. Software Model Checking

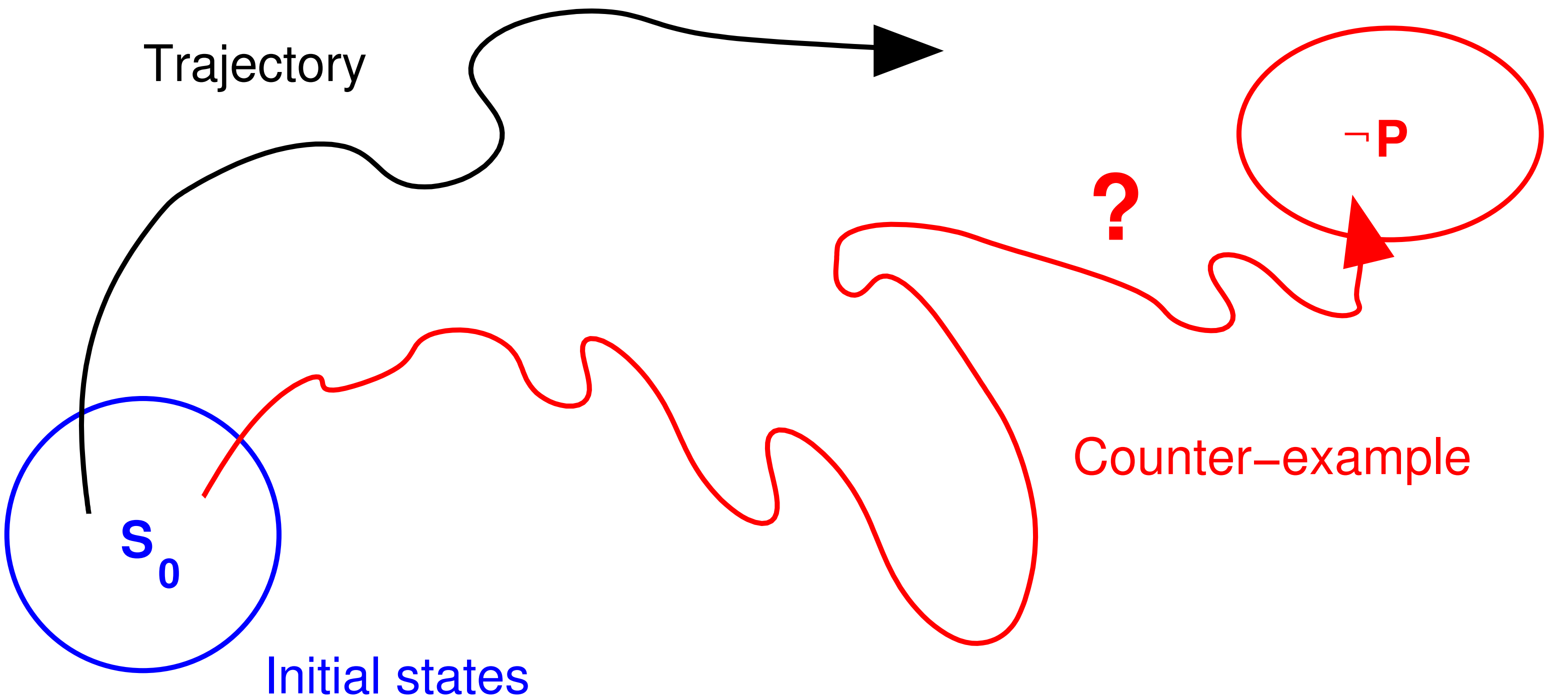
2. Introduction to Java PathFinder:

   (a) Basic usage.
   (b) **`fork`**/**`join`**.
   (c) Deadlock detection.
   (d) **`wait`**/**`notify`**.

Understand JavaPathFinder and its output.

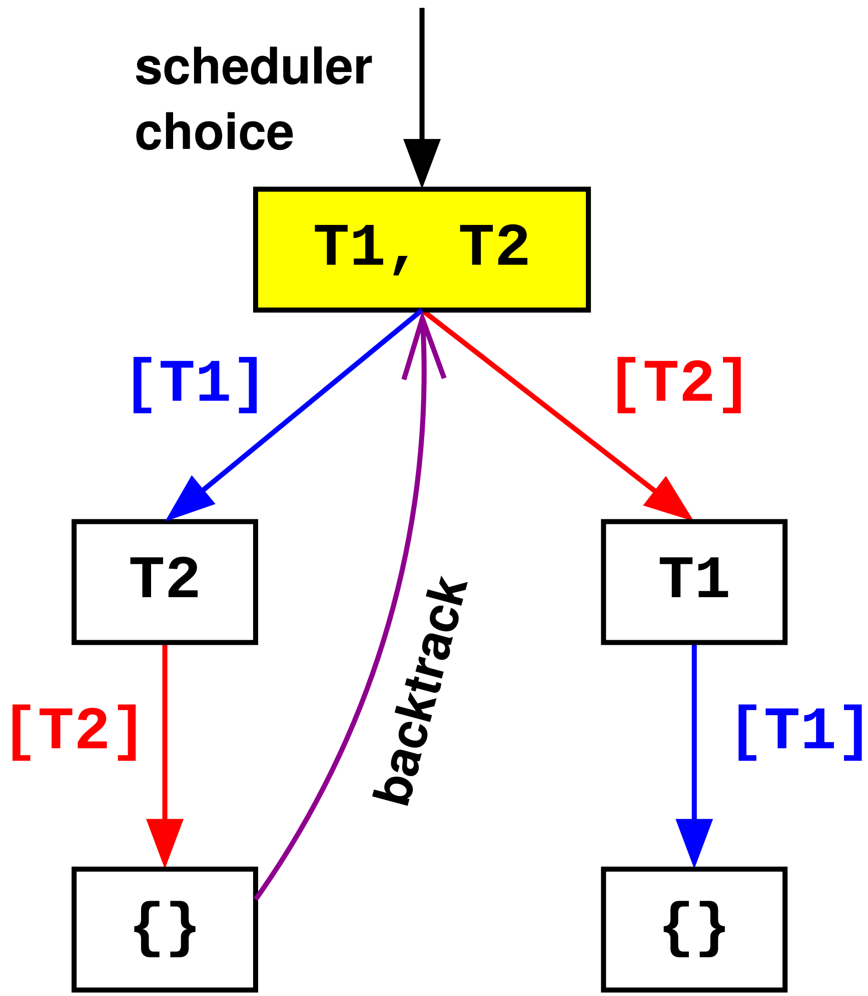
- Furthermore, crash course about thread control and locking in Java.

Model Checking = state space search



- Traditionally applied to specifications, protocols, algorithms.
- Certain types of software (embedded) can be mapped to such model checkers.

Software model checking



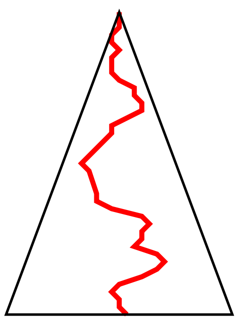
- MC can **backtrack** program execution.
- Explores all possible thread schedules!
- Finds all possible program failures.

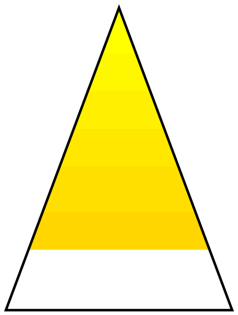
Key limitation: Scalability.

Another limitation: Networking!

Model checking vs. testing

Testing	**Software Model Checking (SMC)**
• **Only one trace.**	• Many (all) traces.
• No rules (but assertions).	• Monitoring rules.
• May miss defects.	• Finds all defects.
• Scalable.	• Resource hungry.

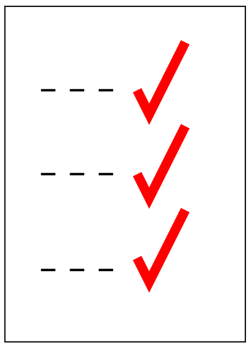




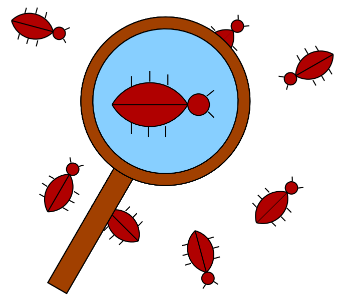
- SMC is a rigorous formal method.
- SMC does not suffer from false positive (like static analysis).
- SMC provides traces (execution history) when it finds a defect.

Heuristic model checking

Full Verification



- Formal verification.
- Prove properties.
- Include all states.
- Not very scalable.



Fault-finding

- Partial verification.
- May miss defects.
- Many, but not all states.
- May find bugs faster.

- Heuristics:
  - Prefer states that likely result in a failure.
  - Helpful with a priori knowledge about defect (deadlock? data race?).
  - Heuristics may be supported by static analysis:
    Rungta, Mercer: A Meta Heuristic for Effectively Detecting Concurrency Errors. HVC 2008.

What is Java PathFinder?

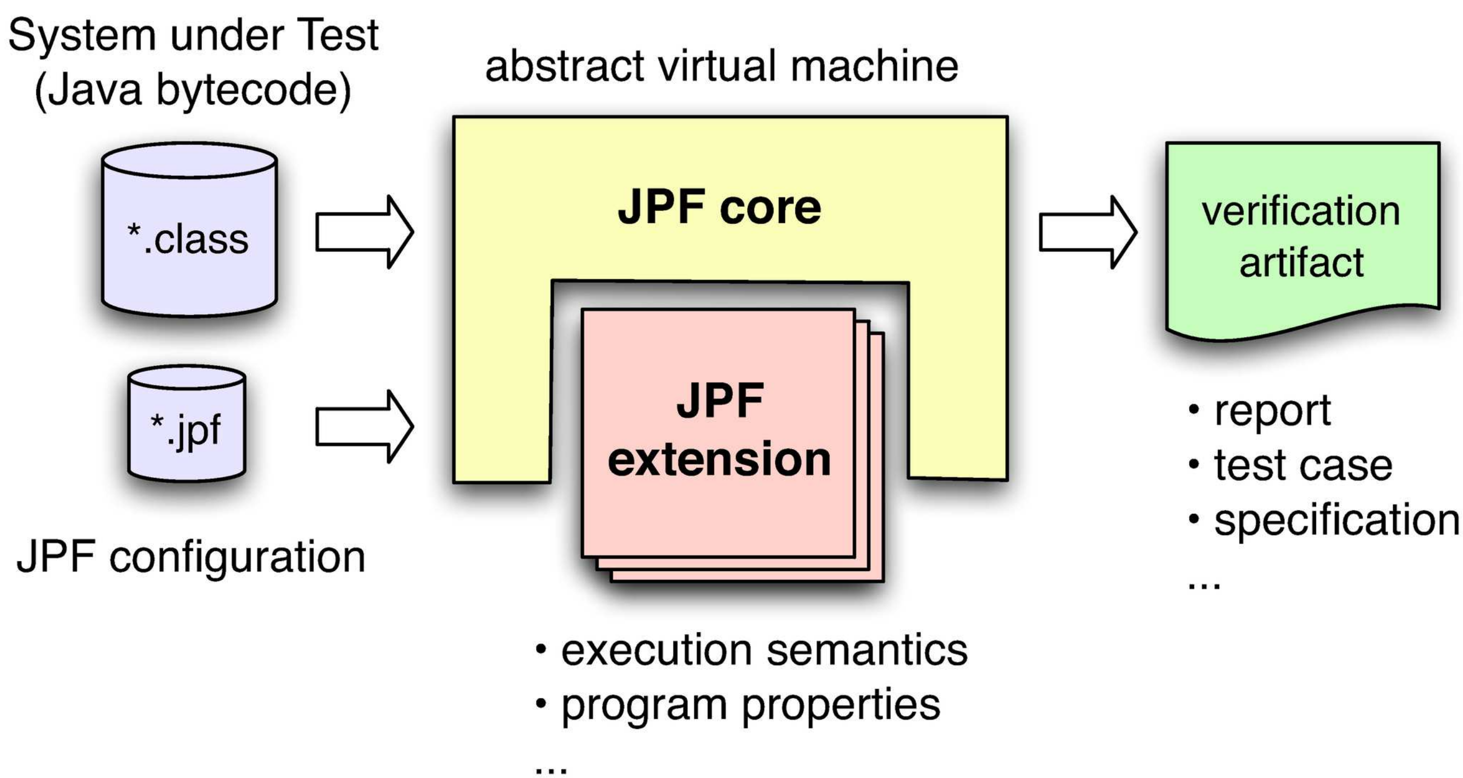


Image source: Java PathFinder tutorial, `http://babelfish.arc.nasa.gov/trac/jpf/attachment/wiki/presentations/start/jpf-graphics.zip`

- Born as software model checker for Java bytecode.
- Evolved into multi-purpose tool framework.

History of Java Pathfinder

- **1999: Project started** at **NASA Ames** as front end to Spin model checker.
- **2000: Reimplementation** as concrete virtual machine for software model checking (concurrency defects).
- **2003:** Introduction of **extension** interfaces.
- **2005: Open sourced** on sourceforge.
- 2008: First participation in Google Summer of Code.
- 2009: Moved to own server, hosting extension projects and wiki.
- **2017: Moved to github.**
- Many collaborators/users:
  - Academic research: > 20 univ. (USA, Canada, Japan, South Africa, Europe).
  - Companies: Fujitsu, other Fortune 500 companies.

Installing Java PathFinder

- You first need to install **Java 8**, **gradle**, and **git.**
- Clone the JPF source code repository:
  `https://github.com/javapathfinder/jpf-core/`
- Run the following commands:

```
mkdir ~/jpf
cd ~/jpf
git clone https://github.com/javapathfinder/jpf-core.git
cd jpf-core
./gradlew
```

- Configuration files have been prepared to be used with JPF 8.

Testing your JPF installation

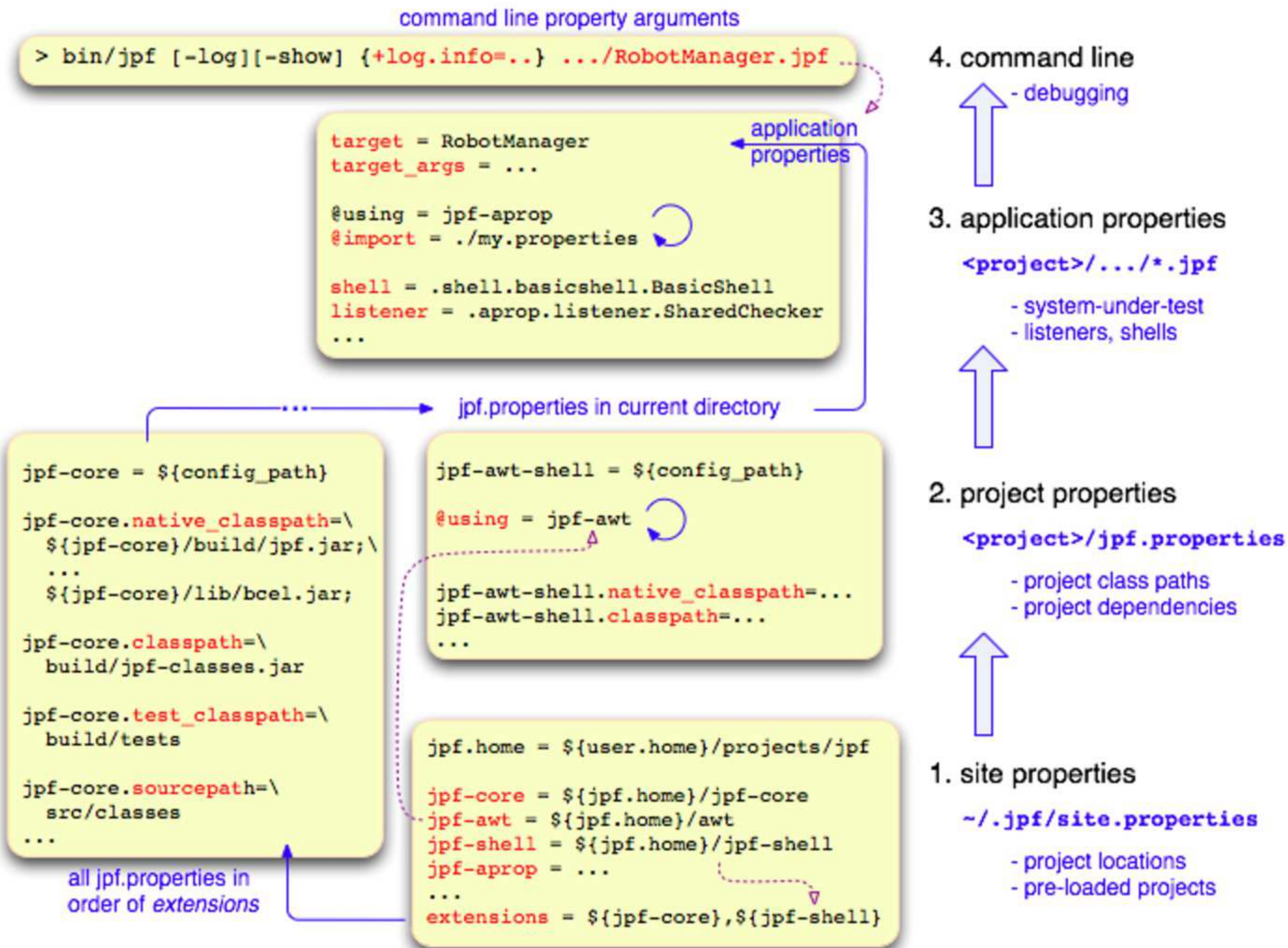
Recommended: Try this before Easter.

The following commands should provide meaningful output:

```
bin/jpf src/examples/HelloWorld.jpf
bin/jpf -version
```

- Tested on Mac OS X and the lab computer's Ubuntu.
- Second command also prints an error message (no file given); you can ignore that as long as you do not see only an error message.

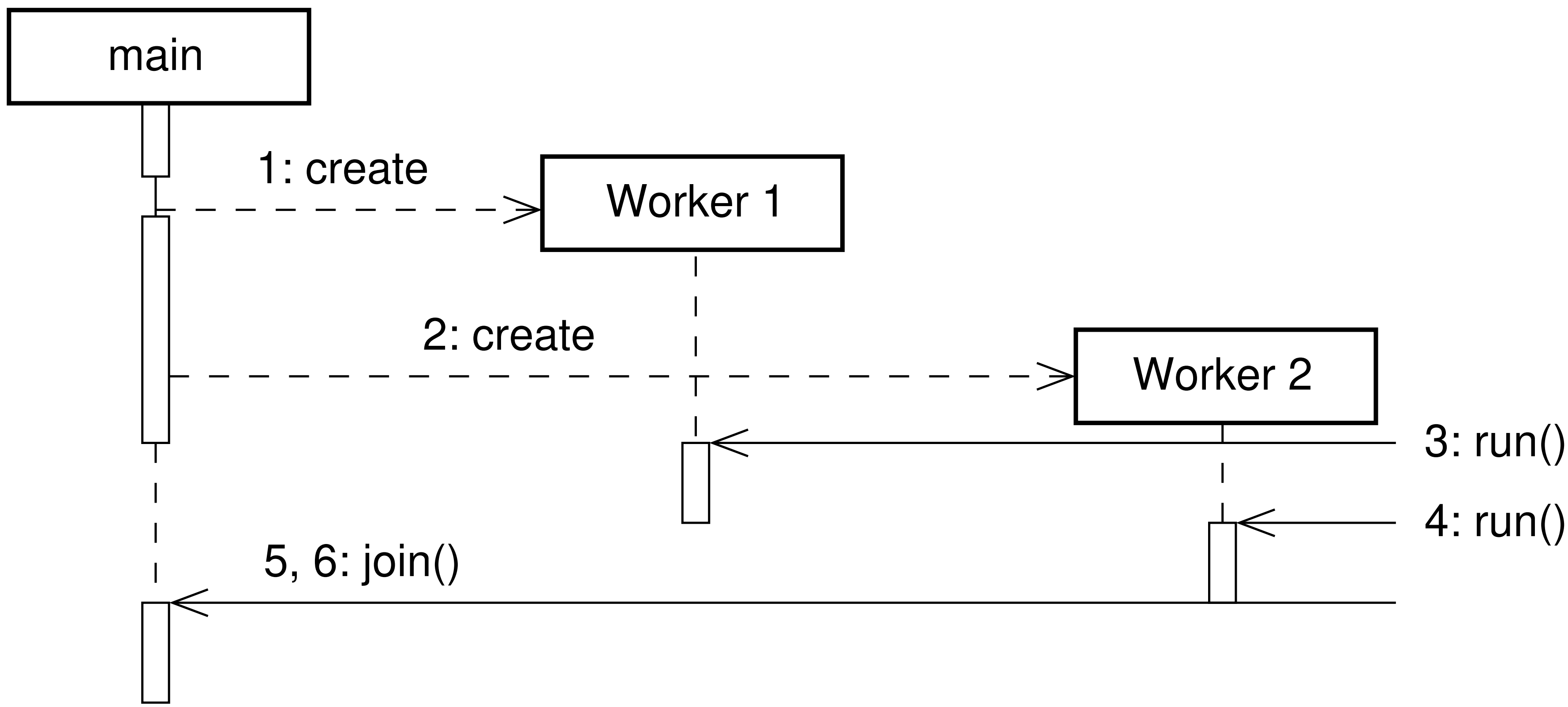
Configuration under JPF 8 (slide from JPF manual)



Example 1: Concurrent „Hello, World!”

```
public class HelloWorld extends Thread {
  public static final void main(String[] args) {
  StringBuffer buf = new StringBuffer();
  HelloWorld hw1 = new HelloWorld(buf, "Hello, ");
  HelloWorld hw2 = new HelloWorld(buf, "World!");
  hw1.start(); // spin off first worker thread
  hw2.start(); // spin off second worker thread
  try {
    hw1.join(); // wait for first worker to finish
    hw2.join(); // wait for second worker
  } catch (InterruptedException e) {
  }
  System.out.println(buf.toString());
  }
}
```

Concurrency in Java

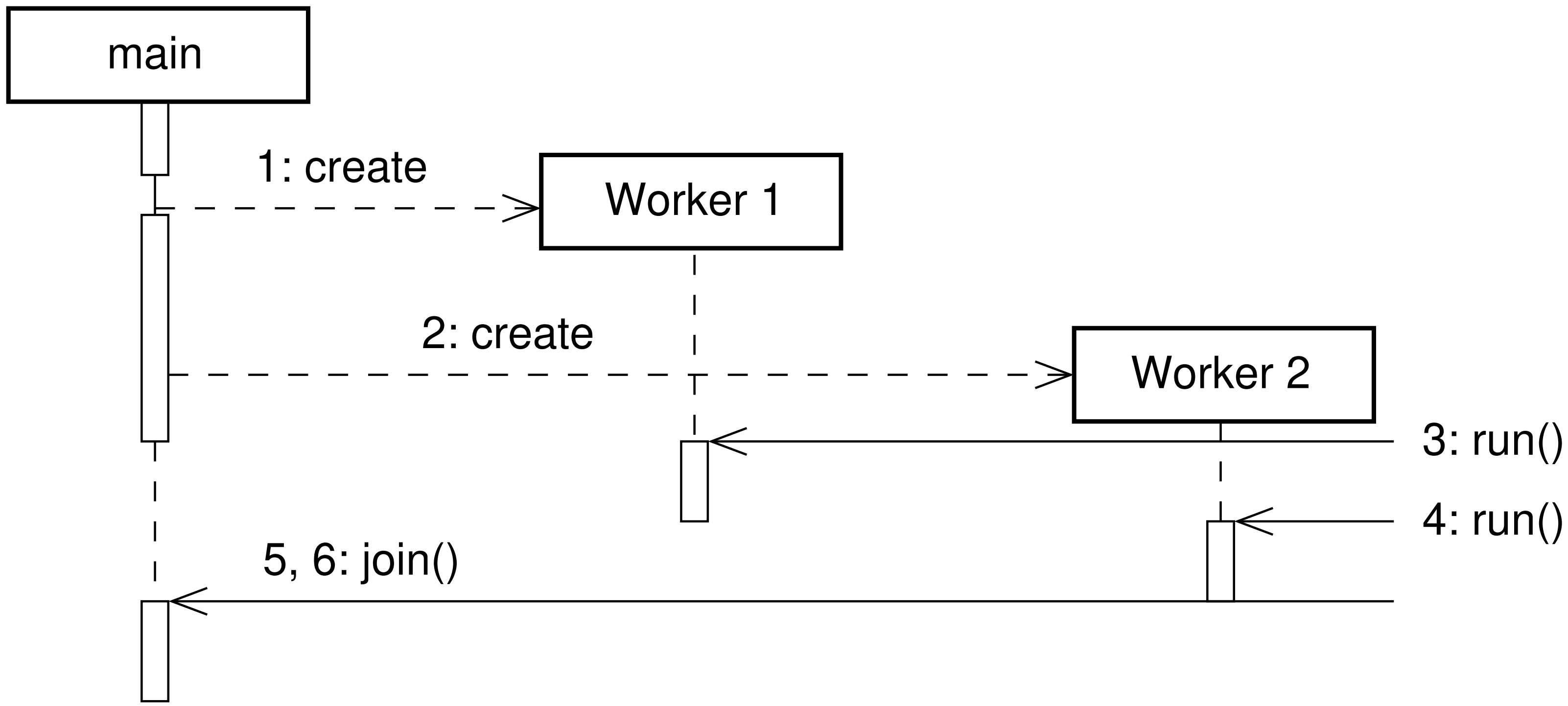


In Java, a child thread is started as follows:

1. Create an instance of class **`Thread`**.
   Alternative: Subclass of **`Thread`**, class implementing interface **`Runnable`**.

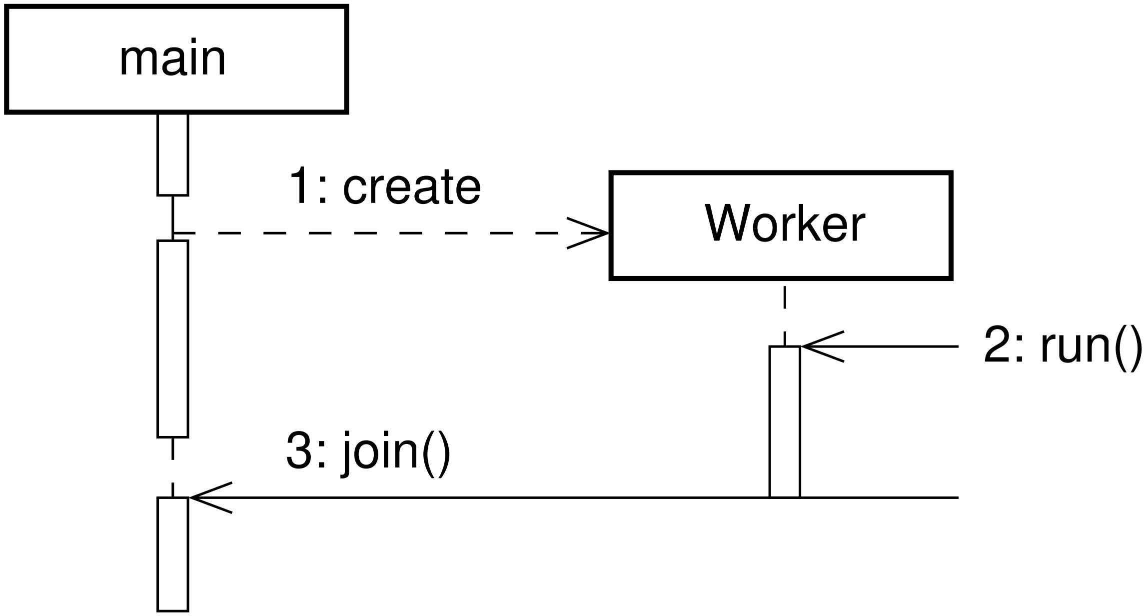
2. Call to **`start`** launches new thread as separate unit of execution.

Concurrency in Java (2)



- Constructor of **`Thread`** instance is initialized from main thread.
- „Payload" (function to be executed) of thread is in **`run`** method.
- Payload is executed from separate thread of execution („in parallel").
- Execution of child thread may be immediate or delayed!

Fork/join parallelism



- `start` „spins off”/”forks” a new thread, executing its `run` method.
- Both main and child thread may run in parallel (given enough CPUs).
- `join` blocks current thread until target thread has terminated.

Fork/join by example

```
hw1.start(); // spin off first worker thread
hw2.start(); // spin off second worker thread
try {
  hw1.join(); // wait for first worker to finish
  hw2.join(); // wait for second worker
} catch (InterruptedException e) {
}
```

- Both child threads run independently.
- Parent waits for first child to terminate, then second one.
- `try`/`catch` block is just „boilerplate” here.
  - If current thread is interrupted while inside `join`, then exception is triggered.
  - This scenario cannot occur in this simple example.

The „payload” of HelloWorld

```
public class HelloWorld extends Thread {
  StringBuffer buf;
  String data;
  HelloWorld(StringBuffer buf, String data) {
    this.buf = buf;
    this.data = data;
  }
  public void run() {
    buf.append(data);
  }
```

- **`run`** appends a string to the (shared) buffer.
- In **`main`**:

  ```
  hw1 = new HelloWorld(buf, "Hello, ");
  hw2 = new HelloWorld(buf, "World!");
  ```

- For expected result: **`run`** methods need to be executed in the right order.

Running HelloWorld in the normal JVM

```
% java HelloWorld
Hello, World!
```

- Nothing special here.

Running HelloWorld in Java PathFinder

```
% jpf HelloWorld
JavaPathfinder v6.0 (rev 616) - (C) RIACS/NASA Ames Research Center
========================= system under test
application: HelloWorld.java
========================= search started: 11/7/11 1:42 PM
Hello, World!
Hello, World!
World!Hello,
Hello, World!
World!Hello,
========================= results
no errors detected
========================= statistics
elapsed time:        0:00:00
states:              new=68, visited=67, backtracked=134, end=5
search:              maxDepth=10, constraints=0
choice generators:   thread=68 (signal=0, lock=11, shared ref=16), data=0
heap:                new=390, released=111, max live=352, gc-cycles=123
instructions:        4466
max memory:          81MB
loaded code:         classes=79, methods=1386
========================= search finished: 11/7/11 1:42 PM
```

Explanation of the output

- Java PathFinder tries out all interleavings.
- Order in which both worker threads run is not restricted (enough)!
- **`hw2`** may run first, resulting in „World! Hello, ".

This is interesting, but how did it happen?

- No properties specified $\rightarrow$ no errors found $\rightarrow$ no explanation.
- We need to add properties to our software!
- Simplest property: safety (assertion).

Automating the verification of the output

- Check value against expected value using assertion.
- Change line with print statement to
  `assert (buf.toString().equals("Hello, World!"));`
- Now we compare the output to the expected value.
- Assertion failure results in program termination (error trace from JPF).

Error trace in JPF

```
~/jpf/jpf-core/bin/jpf HelloWorld.jpf > jpf.log
```

- We need the source path to see the source code statements in the trace.

```
================================ error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.AssertionError
 at HelloWorld.main(HelloWorld.java:25)
================================ trace #1
-------------------------------- transition #0 thread: 0
... // error trace shown here
================================ results error #1:
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
  "java.lang.AssertionError at HelloWorld.main(Hello..."
...
```

How to read the error trace

Thread number	Purpose	Thread name
0	main thread	main
1	add „Hello, " to string buffer	Thread-0
2	add „World!" to string buffer	Thread-1

- Watch out for thread # changes in

```
------------- transition #3 thread: 2
```

- Corresponding thread name of new thread is shown in

```
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet... {Thread-1,>Thread-2}
```

- Number in thread name does not match thread number!
- Reading the trace, **write down summary of thread actions side by side.**

Fixing the example application

```
try {
  hw1.join();
} catch (InterruptedException e) {
}
hw2.start();
try {
  hw2.join();
} catch (InterruptedException e) {
}
```

- One possibility: create two **join** blocks.
- How to describe that change?

Highlighting the change: `diff`

- Automatically describes difference between text files.

- Applied to example source code:

```
% diff hw-assert/HelloWorld.java hw-fixed/HelloWorld.java
19d18
<   hw2.start();
21a21,24
>   } catch (InterruptedException e) {
>   }
>   hw2.start();
>   try {
```

- < refers to first file (old version), > to second file (new version).

Unified `diff`

```
% diff -u hw-assert/HelloWorld.java hw-fixed/HelloWorld.java
--- hw-assert/HelloWorld.java 2010-11-16 13:41:00.000000000 +0900
+++ hw-fixed/HelloWorld.java 2010-11-16 15:48:59.000000000 +0900
@@ -16,9 +16,12 @@
  HelloWorld hw1 = new HelloWorld(buf, "Hello, ");
  HelloWorld hw2 = new HelloWorld(buf, "World!");
  hw1.start();
- hw2.start();
  try {
      hw1.join();
+ } catch (InterruptedException e) {
+ }
+ hw2.start();
+ try {
      hw2.join();
  } catch (InterruptedException e) {
  }
```

- **–** refers to old version, **+** to new version.
- Easier to read and more stable thanks to context (unchanged lines).

Mutual exclusion and locking

- Data race occurs because actions of thread are not atomic.
- Non-interference between actions can be ensured through „critical sections”.
  - Only one thread may enter a critical section at any time.
  - In Java, critical sections are implemented through **locking.**
  - Each critical section (on given data) has to use the same lock.
- **Locks:** „Gate keepers” to blocks of code (critical section).
  - Each lock may only be held by one thread at a given time.
  - If thread tries to obtain lock, then either. . .
    1. It atomically checks for its availability and obtains it, if possible, or
    2. It is suspended if the lock in question is not available.
  - Suspended thread may retry locking at any time.
- **Deadlock:**
  - Several threads are waiting for a resource that never becomes available.
  - „Deadly embrace”: Mutual dependency on resource owned by other threads.

Locking in Java: `synchronized`

- In Java, a thread may obtain a lock on any object instance.

```
Object lock = new Object(); // anything may be a lock
static class Lock {}
lock = new Lock();          // this is just „syntactic sugar”
```

- The keyword **`synchronized`** is used to obtain and release a lock:

```
synchronized (lock) {
  ... // critical section
}
```

- Lock is held inside the scope of **`synchronized`** block.
  - Lock is always obtained/release pairwise, within same method.
  - Lock can be obtain multiple times *(reentrancy).*
    Reentrant operations only affect lock count.

More information about concurrency/locking in Java

Official tutorial:
`http://download.oracle.com/javase/tutorial/essential/concurrency/`

- Covers more detail than what is needed here.

- Recommended sections:
  - Defining and Starting a thread
  - Joins
  - Synchronization
  - Liveness
  - Guarded Blocks

- We will use some of the features covered there in future lectures.
  Please study these sections before the first lab assignment.

Exercise 1, part 1: Dining Philosophers

```
public class DiningPhil {
  static class Fork {
  }
  static class Philosopher extends Thread {
    Fork left;
    Fork right;
    public Philosopher(Fork left, Fork right) {
      this.left = left;
      this.right = right;
      start();              // spin off new thread
    }
    public void run() {
      // think!
      synchronized (left) {
        synchronized (right) {
          // eat!
        }
      }
    }
  }
```

Dining Philosophers: test harness

```
  static final int N = 5;
  public static void main(String[] args) {
    Fork[] forks = new Fork[N];
    for (int i = 0; i < N; i++) {
      forks[i] = new Fork();
    }
    for (int i = 0; i < N; i++) {
      new Philosopher(forks[i], forks[(i + 1) % N]);
    }
  }
}
```

Dining Philosophers: Execution

- Normal JVM is unlikely to encounter deadlock:
  `java DiningPhil`
  # should succeed, unlikely to encounter deadlock

- JPF finds possible deadlock:
  `~/jpf/jpf-core/bin/jpf DiningPhil.jpf`

JPF output: part 1

```
JavaPathfinder v6.0 (rev 616) - (C) RIACS/NASA Ames Research Center
========================= system under test
application: DiningPhil.java
========================= error #1
gov.nasa.jpf.jvm.NotDeadlockedProperty
deadlock encountered:
  thread index=1,name=Thread-1,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
  thread index=2,name=Thread-2,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
  thread index=3,name=Thread-3,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
  thread index=4,name=Thread-4,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
  thread index=5,name=Thread-5,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
```

- **`lockCount`/`suspendCount`** can be ignored (refers to reentrant lock usage).

JPF output: part 2

```
============================== trace #1
------------------------------ transition #0 thread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {>main}
      [2895 insn w/o sources]
  DiningPhil.java:50      : Fork[] forks = new Fork[N];
  DiningPhil.java:51      : for (int i = 0; i < N; i++) {
  DiningPhil.java:52      : forks[i] = new Fork();
  DiningPhil.java:23      : static class Fork {
...
  DiningPhil.java:31      : public Philosopher(Fork left, Fork right) {
      [188 insn w/o sources]
  DiningPhil.java:32      : this.left = left;
  DiningPhil.java:33      : this.right = right;
  DiningPhil.java:34      : start();
      [1 insn w/o sources]
```

- Summary of this transition:
  create fork/philosopher instances, start threads.

JPF output: part 3

```
============================== snapshot #1
thread index=1,name=Thread-0,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
  owned locks:DiningPhil$Fork@144
  blocked on: DiningPhil$Fork@145
  call stack:
 at DiningPhil$Philosopher.run(DiningPhil.java:40)
thread index=2,name=Thread-1,status=BLOCKED,
priority=5,lockCount=0,suspendCount=0
  owned locks:DiningPhil$Fork@145
  blocked on: DiningPhil$Fork@146
  call stack:
 at DiningPhil$Philosopher.run(DiningPhil.java:40)
...
```

- After all 10 transitions, thread status of remaining 5 threads is shown.
- All threads are blocked (waiting on lock) in this case: **deadlock!**

JPF output: part 4

```
================================= results
error #1: gov.nasa.jpf.jvm.NotDeadlockedProperty
  "deadlock encountered:   thread id=1,name=Thread-1,..."
================================= statistics
elapsed time:       0:00:01
states:             new=101, visited=136, backtracked=224, end=11
search:             maxDepth=16, constraints=0
choice generators: thread=100 (signal=0, lock=25, shared ref=0), data=0
heap:               new=387, released=1065, max live=376, gc-cycles=237
instructions:       7652
max memory:         81MB
loaded code:        classes=83, methods=1322
================================= search finished
```

Exercise: Dining Philosophers

1. **Create summary information of all thread actions:**

Trans. \ Thread name/ID	main 0	Thread-0 1	... ...	Thread-4 5
0	create fork/ phil. instances, launch threads			
1–2		obtain lock *x* try to obtain lock *y*		

   Complete table (and make sure to set correct lock IDs for *x*, *y*). Lock ID should uniquely identify each lock.

2. **Create lock dependency graph** (for remaining 5 threads). Nodes: thread ID + locks held by thread. Edge $a \xrightarrow{x} b$: Thread a wants to obtain lock x held by thread b.

3. Propose possible **fix** for **program**, create „patch" (diff output). Revised version of program should pass through JPF with no errors.

Lab exercise

1. Summary of thread actions: plain text or PDF (print to PDF from spreadsheet), scan from paper.

2. Lock dependency graph: PDF or scan from paper.

3. Copy file before editing: **`cp DiningPhil.java DiningPhil.java.orig`**.
   Submit diff output:
   `diff -u DiningPhil.java.orig DiningPhil.java > DiningPhil.patch`.

Requirements for revised philosophers

- Each philosopher still uses two forks together.
- Lock ordering may be changed.
- Number of philosophers = number of forks = N.
- Additional (non-Fork) locks may be used (although not necessary).
- Number of philosophers/forks remains at N = 5
  (JPF is able to handle resulting state space).
- No „magic numbers" (constants other than 0, 1, N).
  N is defined exactly once in the code (as in existing version).
  Use `N - 1`, `... % N`, etc.
  Code should still work when N is changed.
- **Hint:** If you need more than 5–10 extra lines of code,
  then your solution is probably more complicated than necessary.

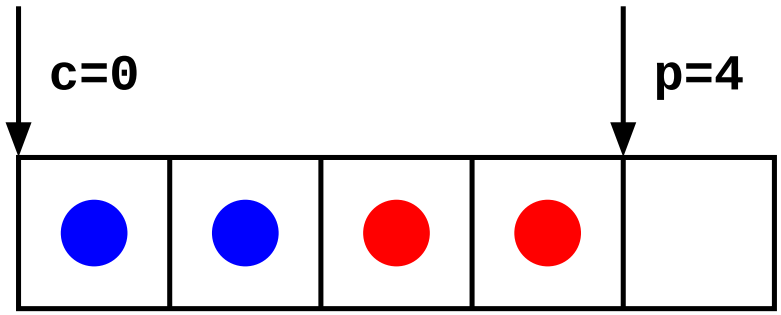
Shared conditions

- Locking can protect critical sections against concurrent access.
- Major building block for creating concurrent algorithms.
- Other operations required for more complex algorithms.

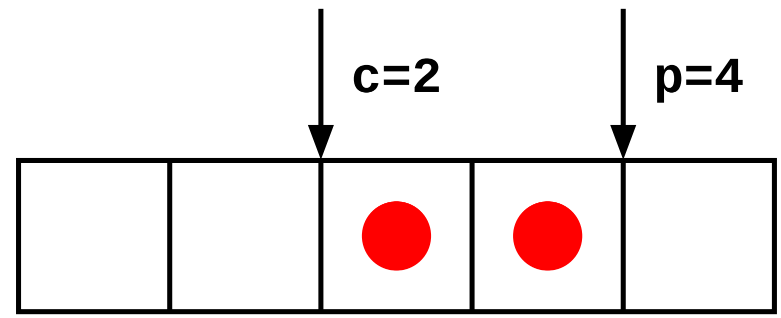
Producer/Consumer (Queue)

- Queue that allows concurrent data access to fixed-size buffer.
- Multi-element **`put`** blocks when queue is full (would overflow on put).
- Multi-element **`remove`** blocks when queue does not contain enough elements.
- Two indices:
  - Reader index **`c`** („consumer”).
  - Writer index **`p`** („producer”).
  - Both **`c`** and **`p`** may wrap around queue size (modulo).
  - Wrap-around: invariant $c + size \geq p$ (even after element insertion).

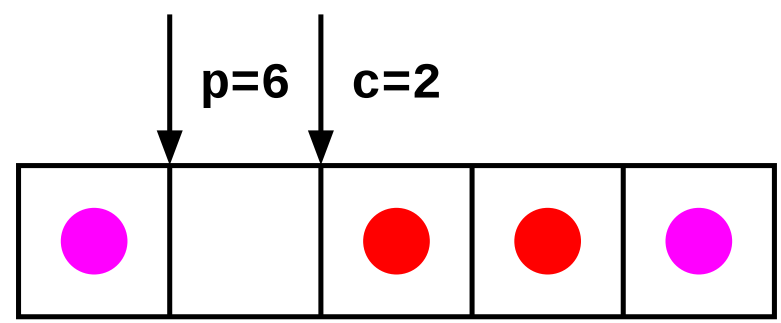
Queue implementation



After 2 put (2 elements each)



2 put, 1 remove



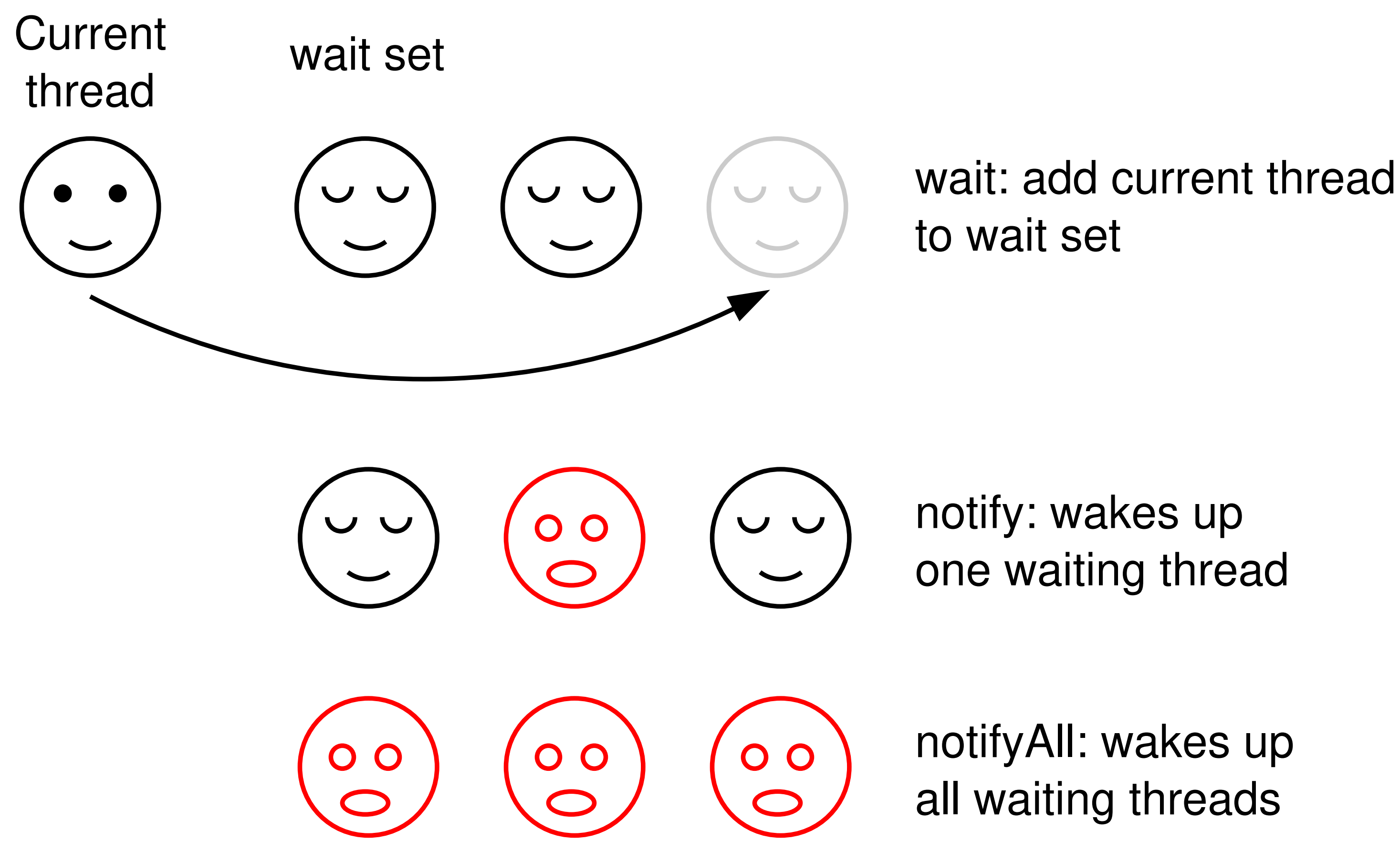
2 put, 1 remove, 1 put

Queue source code: `remove`

```
public synchronized void remove(byte[] storage) {
  int len = storage.length;
  waitForData(len);
  for (int i = 0; i < len; i++) {
    storage[i] = data[c++ % size];
  }
  /* index reduction to prevent index overflow */
  if (c >= size) {
    assert (p >=c);
    p -= size;
    c -= size;
  }
  notifyAll(); // wake up waiting producers
}
```

- **`synchronized`** ensures atomic behavior of operation.
- **`waitForData`** blocks until at least **`len`** elements are available.
- **`notifyAll`** signals any threads blocking on **`put`** (producers) that storage has increased after removal.

Shared conditions: `wait/notify`



- **`wait`** allows a thread to wait until another thread fulfills a (pre-)condition.
- **`notify`**/**`notifyAll`** signals waiting threads that condition is fulfilled.

`wait`/`notify` usage

- Lock on **`x`** must be held when **`x.wait`** or **`x.notify`** is called.

- **`x.wait`**:

  1. Releases lock on **`x`** (lock release not visible in source code).
  2. Puts current thread to sleep.

- **`x.notify[All]`**:

  1. Enables one or all threads to wake up.
  2. Lock on **`x`** is still held until **`synchronized`** block is exited!
  3. After release of lock **`x`**, newly awakened threads try to re-obtain lock **`x`** when scheduled.

- Even if all threads are woken up together,
  only one thread at a time, may take lock and continue!

- Condition waited on may no longer hold when awakened thread continues.
  This is a very frequent source of programming errors!

Waiting for something to happen

Assuming lock on **`this`** is held, ***`await(condition)`*** becomes six-liner in Java:

```
while (!condition) {
  try {
    wait();
  } catch (InterruptedException e) {
  }
}
```

- Condition needs to be re-checked **before and after `wait`**.
- Monotonic conditions may be checked only before (using **`if`** instead of **`while`**).
- Inability of other threads to establish condition will result in deadlock (sometimes livelock).

Establishing a condition

Check condition

```
// wait for condition
synchronized (instance) {
  while (!condition) {
    try {
      instance.wait();
    } catch (Int.Exception e) {
    }
  }
  assert (condition);
  // holds unless data race present
}
```

Establish condition

```
// establish condition,
// notify waiting threads
synchronized (instance) {
  // always use locking when
  // modifying shared data
  condition = true;
  notifyAll();
}
```

- **`condition`** is usually not a boolean variable but an expression.
- Any data access that modifies part of **`condition`** (expression) needs synchronization!
- Synchronization may be done outside current method (by caller).

Queue source code: `waitForData`

```
private void waitForData(int len) {
  assert (len <= size);
  while (c + len > p) {
    try {
      wait();
    } catch (InterruptedException e) {
    }
  }
  assert (p >= c + len) :
    "p >= c + len failed: p = " + p + ", c = " + c + ", len = " + len;
  assert (p - size <= c) :
    "p - size <= c failed: p = " + p + ", c = " + c;
}
```

- **`while (!condition)`** $\rightarrow$ condition holds at end of **`while`**/**`wait`** loop.
- $\neg(c + len > p) \rightarrow c + len \leq p \rightarrow p \geq c + len$
- Lock must be held outside this helper function;
  then **`waitForData`** ensures availability of len elements.

Queue source code: `put`

```
/** Atomic put for multiple elements. Blocks until space available. */
  public synchronized void put(byte[] items) {
    int len = items.length;
    assert (len <= size);
    // wait for space to become available
    while (p + len > c + size) {
      try {
        wait();
      } catch (InterruptedException e) {
      }
    }
    assert (p + len <= c + size) :
      "p + len <= c + size failed: p = " + p + ", c = " + ", len = " + len;
    for (int i = 0; i < len; i++) {
      data[p++ % size] = items[i];
    }
    assert (p <= c + size) :
      "p <= c + size failed: p = " + p + ", c = " + c;
      notifyAll(); // wait up waiting consumers
  }
```

Producer/consumer dance

Producer		**Consumer**
add data		
notify (cons.)	$\rightarrow$	wait for data
$\uparrow$		$\downarrow$
wait for space	$\leftarrow$	remove data
		notify (prod.)

- Temporally, a producer must come first (as consumer would block).
- In program code, condition needs to be checked first.
- In Java, notifications *cannot* be tailored to producer or consumer threads (either all waiting threads, or one at random, no pre-determined subset).
- **`notify`** may wake up the „wrong" thread $\rightarrow$ use **`notifyAll`**, as we have threads waiting on *different* conditions.

Simple test bench: `ProdCons`

```
public class ProdCons {
  ... // declaration of constants
  static Queue q;
    static class Producer extends Thread {
    byte[] data;
    Producer(int i) {
      data = new byte[]{(byte)i, (byte)(i + DELTA)};
    }
    public void run() {
      q.put(data);
    }
  }
  ...
}
```

- Idea: test insertion of two elements at a time, $\langle i, i+\delta \rangle$.
- If retrieval of $\langle i, j \rangle$ is atomic, then $j = i + \delta$.

Test bench: Consumer and `main`

```
static class Consumer extends Thread {
  public void run() {
    byte[] result = new byte[2];
    q.remove(result);
    System.out.println(result[0] + ", " + result[1]);
    assert (result[1] - result[0] == DELTA);
  }
}

public static void main(String[] args) {
  q = new Queue(Q_SIZE);
  for (int i = 0; i < N; i++) {
    new Producer(i).start();
    new Consumer().start();
  }
}
```

- Result is printed on screen so JPF's state space search is visible.

Can we optimize the code?

- Obviously, at least one entry has to be produced before anything can be read.
- So any consumer thread has to wait until data is available.
- Should we always wait before checking the condition?
- JPF says no!

Thread number	Purpose	Thread name
0	main	main
1	Producer 1	Thread-1
2	Consumer 1	Thread-2
3	Producer 2	Thread-3
4	Consumer 2	Thread-4

Problem: Lost signal

- Data may already be available when producer thread comes first.
- No problem if more producers or consumers arrive and send redundant signal.
- Deadlock if
  1. No more producers (queue is full or no producers available).
  2. No more consumers (which would pass check and send signal).
- Signal from extra consumers would be redundant, mask possible deadlock.

Large-scale stress testing likely to produce redundant signals.

- Behavior may range from inefficiency to actual deadlock.
- Hard to diagnose with traditional testing.

Error trace for lost signal

Trans.	main 0	Thread-1 1	Thread-2 2	Thread-3 3	Thread-4 4
0–6	init. data, launch thr.				
7		add data notifyAll			
8			try to remove data wait		
9				try to add data wait	
10					try to remove data wait

- Signal is „lost” because consumer thread (#2) is too late to catch it.
- Checking the condition before calling **`wait`** resolves this problem.

Exercise1, part 2: `wait/notify` usage; test in `ProdCons`

- In `queue-notify`, `notifyAll` in `Queue.java` is replaced with `notify`.
- JPF now reports deadlock for `ProdCons` (`N` = 2, `Q_SIZE` = 2), 16 transitions.
  `jpf ProdCons.jpf`

Hint:

- **Call to notify does not imply immediate signal delivery!**
- Signal delivery is shown by thread progressing past `wait` call:

```
-------------------------- transition #X thread: A
  ...
  Queue.java:XX           : wait(); // DOES NOT GET SIGNAL
-------------------------- transition #Y thread: B
  Queue.java:XX           : notify();
  ...                       // release lock, continue
-------------------------- transition #Z thread: C
  Queue.java:XX           : wait();
  Queue.java:XX           : }       // GETS SIGNAL
```

Tasks for lab exercise

1. **Analyze error trace, create summary information (table) for trace.**
   Pay close attention to each **`wait`**/**`notify`** call, and received signals.
   - Which signal allows a thread to progress past the enclosing **`while`** loop?
   - Which signal is caught by a thread that *cannot* progress past the loop?
   - Would another thread be able to continue if it received that signal?

2. The error trace obviously shows that the program is incorrect.
   Yet, another test program called **`QueueTest`** does not reveal any problem.
   Study `QueueTest.java`; why can it not cause this particular failure?

Summary

- Simple testing may miss subtle concurrency defects.
- Java PathFinder analyzes all possible thread schedules.
  - May uncover hard-to-find defects.
  - Requires properties to be verified (assertions).
- Error trace may be difficult to read.
  - Several threads involved, much detail.
  - Create summary by writing crucial thread actions side by side.